General purpose graphics-processing-unit implementation of cosmological domain wall network evolution

J. R. C. C. C. Correia[1,2,*] and C. J. A. P. Martins[1,3,†]

[1]*Instituto de Astrofísica e Ciências do Espaço, Universidade do Porto, CAUP, Rua das Estrelas, 4150-762 Porto, Portugal*
[2]*Departamento de Física, Universidade de Aveiro, Campus Universitário de Santiago, 3810-193 Aveiro, Portugal*
[3]*Centro de Astrofísica da Universidade do Porto, Rua das Estrelas, 4150-762 Porto, Portugal*

Topological defects unavoidably form at symmetry breaking phase transitions in the early universe. To probe the parameter space of theoretical models and set tighter experimental constraints (exploiting the recent advances in astrophysical observations), one requires more and more demanding simulations, and therefore more hardware resources and computation time. Improving the speed and efficiency of existing codes is essential. Here we present a general purpose graphics-processing-unit implementation of the canonical Press-Ryden-Spergel algorithm for the evolution of cosmological domain wall networks. This is ported to the Open Computing Language standard, and as a consequence significant speedups are achieved both in two-dimensional (2D) and 3D simulations.

I. INTRODUCTION

Topological defects form at cosmological phase transitions, as a consequence of the Kibble mechanism [1]. If they are stable or sufficiently long-lived they will be present in the more recent universe, as fossil relics of its earlier stages, leading to a plethora of astrophysical signatures [2]. The conceptually simplest way of studying the highly nonlinear evolution of defect networks is by thermodynamic analytic modeling. This is based on an idea of Kibble [3], and the current state of the art is the velocity-dependent one-scale model [4,5]. However, just as in standard thermodynamics there are parameters (such as Boltzmann's constant) which cannot be determined *ab initio* but must be experimentally determined, so analytic models for defect evolution include model parameters which must be determined in high-resolution numerical simulations—thereby calibrating the model.

For the simplest defect model, domain walls from a single scalar field, the canonical approach to field theory simulations is the Press-Ryden-Spergel (PRS) algorithm [6], and state of the art simulations and analytic model calibration are described in [7,8]. This WALLS code has been used as a benchmark for central processing units (CPU) and Intel Xeon Phi coprocessors [9]. For the more commonly studied case of cosmic strings there are several Goto-Nambu [10–14] and field theory codes [15–17]. There are also implementations for monopoles [18], semilocal strings [19], and non-Abelian defects [20,21]. All of these are optimized for standard CPUs, either with shared or distributed memory architectures.

Recent progress in cosmic microwave background [22] and gravitational wave detection [23] highlights how some of these scenarios can be constrained by high-resolution data. However, they also show that the current bottleneck is the lack of efficient and accurate high-resolution simulations of defect networks that can be used as templates for robust statistical analysis. This will be an even bigger problem for next-generation

facilities such as CORE [24] and LISA [25]: the number and resolution of the required simulations eventually require prohibitive amounts of time or hardware costs. It is therefore important to exploit recent hardware and software advances that yield gains in efficiency of these codes. This work is a step in this direction: we present a first implementation of the PRS algorithm for domain walls on general purpose graphical processing units (GPGPUs).

II. DOMAIN WALLS AND THE PRS ALGORITHM

Domain walls arise whenever a discrete symmetry is broken during a phase transition. The simplest toy model describing wall networks stems from the Lagrangian density of a scalar field ϕ,

$$\mathcal{L} = \frac{1}{2}\phi_{,\mu}\phi^{,\mu} - V_0\left(\frac{\phi^2}{\phi_0^2} - 1\right)^2, \tag{1}$$

where the quartic potential $V(\phi)$ has two degenerate minima (and hence the model's vacuum manifold is comprised of two disconnected regions). The equations of motion in a Friedmann-Robertson-Walker universe is obtained by standard variational techniques, leading to

$$\frac{\partial^2\phi}{\partial\eta^2} + \alpha\left(\frac{d\ln a}{d\ln\eta}\right)\frac{1}{\eta}\frac{\partial\phi}{\partial\eta} - \nabla^2\phi = -\alpha^\beta\frac{\partial V}{\partial\phi}, \tag{2}$$

where a is the scale factor and η is the conformal time (related to physical time t by $d\eta = dt/a$). The exact equations of motion have $\alpha = \beta = 2$, but one can show that the numerically more convenient case where the walls maintain constant comoving thickness (corresponding to $\beta = 0$) still satisfies the appropriate momentum conservation laws provided one simultaneously uses $\alpha = 3$. This is the key insight behind the PRS algorithm [6].

This equation can be discretized [6] and the evolution of walls is then described by a first-order (with respect to time) Crank-Nicholson, second-order staggered leap-frog scheme, comprised of three different (embarrassingly parallel) steps

*Jose.Correia@astro.up.pt
†Carlos.Martins@astro.up.pt

which in the 2D case can be written

$$(\nabla^2\phi)_{i,j} = \phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j}, \quad (3)$$

$$\dot{\phi}_{i,j}^{n+1/2} = \frac{(1-\delta)\dot{\phi}_{i,j}^{n-1/2} + \Delta\eta(\nabla^2\phi_{i,j}^n - \partial V/\partial\phi_{i,j}^n)}{1+\delta}, \quad (4)$$

$$\phi_{i,j}^{n+1} = \phi_{i,j}^n + \Delta\eta\dot{\phi}_{i,j}^{n+1/2} \quad (5)$$

(with a straightforward extension to the 3D case), where the damping term δ is given by the expression

$$\delta = \frac{1}{2}\alpha\frac{\Delta\eta}{\eta}\frac{d\ln a}{d\ln\eta}. \quad (6)$$

In order to characterize wall network evolution, two diagnostic quantities are used. The first is the comoving wall area per unit volume (akin to a density),

$$\rho = \frac{A}{V} = \int \boldsymbol{n}\cdot d\boldsymbol{A} = \Delta A\sum_{\text{links}}\delta_{\pm}\frac{\nabla\phi}{|\phi_{,x}| + |\phi_{,y}| + |\phi_{,z}|}. \quad (7)$$

To calculate the area one finds neighboring points where the field changes sign (links), corresponding to an energy concentration associated with the $\phi = 0$ local maximum of the field potential. If a link crosses a wall $\delta_{\pm}$ equals unity; otherwise, it vanishes. This has been shown to be a robust method to calculate the area [26]. The second quantity is the square of the product of the average (root-mean squared) velocity v of the network and the corresponding Lorentz factor. This can be calculated from the sum of the ratio between the kinetic and potential energy of each wall (respectively denoted E_k and V),

$$(\gamma v)^2 = \frac{1}{2N}\sum_{\text{walls}}\frac{E_k}{V(\phi)}, \quad (8)$$

where $\gamma = 1/\sqrt{1-v^2}$ is the Lorentz factor, and the sum is over the number N of grid points containing walls, as identified in the previous step.

It is known from analytic arguments [1,2], confirmed with high-resolution simulations [7,8], that wall networks in a universe whose scale factor grows as a power law (such as the radiation or matter dominated eras) allowed to evolve for a sufficiently long dynamical range will reach an attractor linear scaling solution which numerically corresponds to

$$\rho \propto \eta^{\mu}, \quad \gamma v \propto \eta^{\nu}, \quad (9)$$

where $\mu = -1$ and $\nu = 0$. For simulations with a smaller dynamic range this asymptotic regime may not be reached, which can be identified by a dependence of the exponents μ and ν on the box size [27,28]. The purpose of this paper is to present a parallel implementation of the PRS algorithm for two-dimensional (2D) and 3D domain walls which runs on GPGPUs, using the behavior of these two quantities to validate the implementation.

III. IMPLEMENTATION, SPEEDUPS, AND ERROR ANALYSIS

Our GPGPU implementation of wall network evolution uses the Open Computing Language (OpenCL) 1.2 framework as specified by the Khronos Consortium [29] and implemented

by Apple, Inc., and was developed and tested on a machine equipped with a Radeon R9 M395 Graphics Processing Unit, possessing 28 compute units clocked at 834 MHz, and 2048 MB total video memory clocked at 1365 MHz. On the same machine, the sequential reference version of the same code ran on a Intel i5 6600k with 3.3 Ghz core clock (can boost to 3.9 GHz) and 8192 MB of system memory (clocked at 1867 MHz).

In OpenCL, applications are subdivided in data-parallel functions named kernels, which are to be compiled at run-time (Just In Time compilation). Each step of the PRS algorithm corresponds to a kernel, and so do the velocity and density calculations, with a separate kernel for the sums. These kernels execute in order, one time step at a time. For the sum reduction kernel, we use the scalar version of the kernel in [30]. The reason for not using the vector one (where instead of using vector datatypes like *float4*, one would use *float*, for instance), is that the preferred vector width[1] of the device in question is, for both double and floating point types,

    CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT: 1
    CL_DEVICE_PREFERRED_VECTOR_WID-
TH_FLOAT: 1
    CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE: 1
    CL_DEVICE_PREFERRED_VECTOR_WID-
TH_DOUBLE: 1

so it is equivalent to use either kernel. The sum reduction kernel computes a partial sum for each local memory[2] patch, and all partial sums are transferred back to the host side, summed and written to disk. The only role of the CPU is to sum the partial sums which result from the calculation of the velocity and the density. As a small optimization we use two queues running asynchronously with respect to each other, ensuring overlap between execution of compute kernels and data transfer operations.

Our code is compatible with both double and single precision, though it should be noted that consumer facing graphics cards usually have much lower peak operations per second and as such there is a severe speed penalty in utilizing double precision (for AMD cards based on the Graphics Core Next architecture this varies between 1/2 and 1/16 of peak single precision operations per second [31,32]). This expectation is confirmed by our analysis, summarized in Fig. 1,

---

[1] The OpenCL compiler automatically packs the preferred number of work items or threads into single-instruction-multiple-data lanes and henceforth takes advantage of the native vector width. The native width is the number of elements a vector arithmetic logic unit can process at once.

[2] The OpenCL memory model describes several types of memory: global (which on a graphics card corresponds to video memory), local (a fast-access cache on each compute unit), constant (technically part of video memory as well, but constant), and private (memory bound to each work item or thread). There is a tendency in this code to try to utilize local memory whenever possible, due to its faster access times. We note that we still need to port two kernels to use local memory: the Laplacian and the density kernel. Concatenation of kernels should also follow suit, in order to reduce the number of times one copies to and from memory.
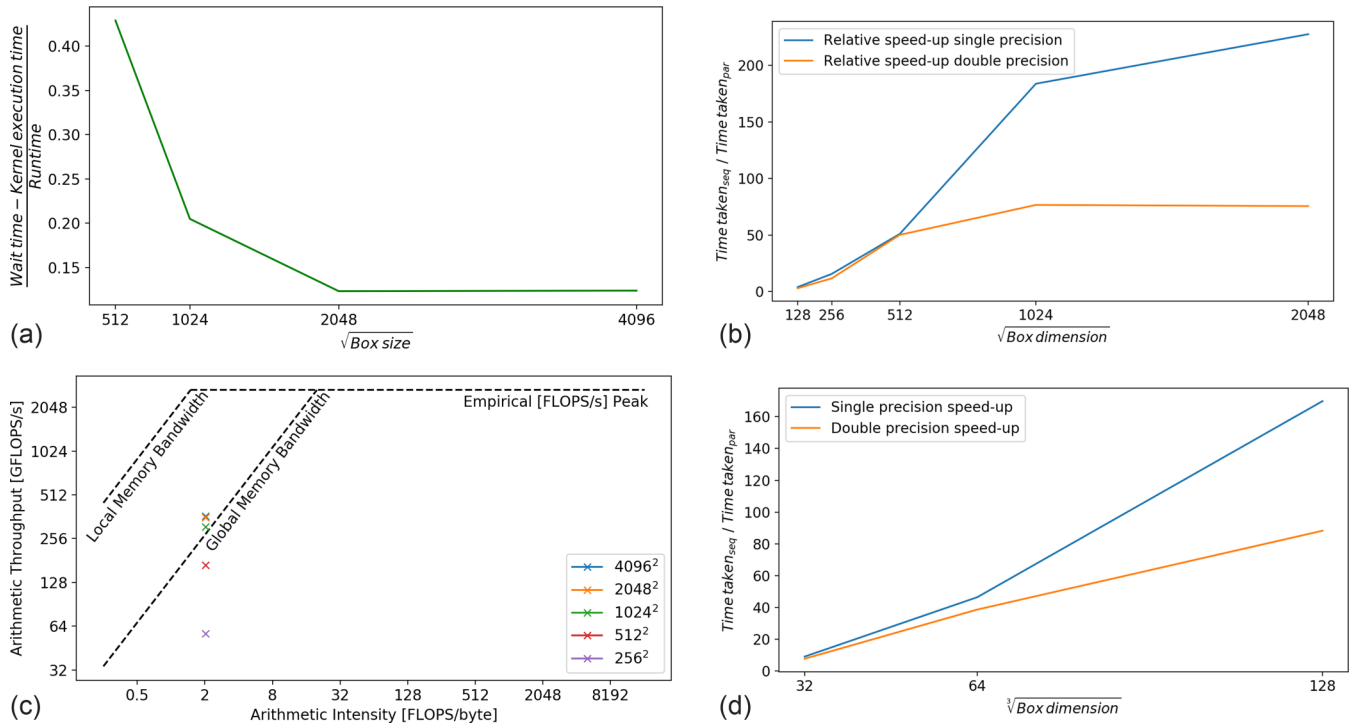
FIG. 1. Top left: estimate of the time wasted in data transfer, or how good the overlap between compute and data transfer is, for different box sizes. Bottom left: roofline model for the 2D implementation. Right: relative speed-up of the parallel version when compared to the sequential one, for both single (blue) and double (orange) precision, for 2D (top) and 3D (bottom) simulations.

which also highlights the large relative speedup of going parallel provided the box size is large enough to fully exploit a GPU.

We also note a few characteristics of the implementation. The fields are represented in memory using buffer data (linear contiguous), and the number of threads (work items) spawned
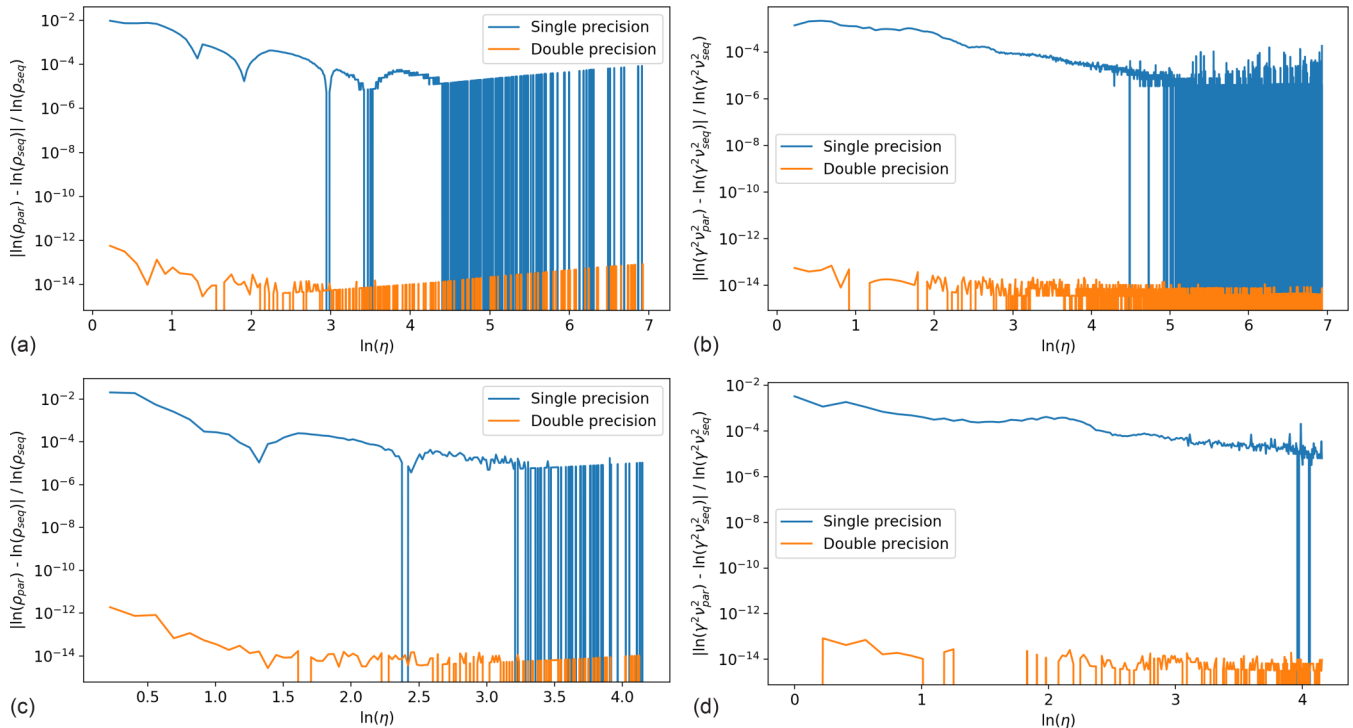

FIG. 2. Relative error between sequential and parallel code implementations, with 2048^2 boxes (top panels) and 128^3 (bottom panels), for both single (blue) and double precision (orange), for the wall density (left panels) and the velocity (right panels).

TABLE I. Scaling exponents μ and ν (with 1σ statistical errors) for single and double precision runs, calculated using the points beyond $\log(\eta) = 2.58$ for both 2048^2 and 128^3 simulations.

	μ	ν
2048^2		
Single precision	-0.9381 ± 0.0003	-0.0374 ± 0.0005
Double precision	-0.9381 ± 0.0003	-0.0374 ± 0.0005
128^3		
Single precision	-0.956 ± 0.003	-0.034 ± 0.006
Double precision	-0.905 ± 0.002	-0.025 ± 0.004

are always equal to the number of points in a box. The OpenCL compiler (and the underlying hardware) handle the distribution of threads automatically. The implementation has low arithmetic intensity, and seems mostly compute bound (when taking local memory bandwidth into account; see roofline model in Fig. 1). From AMD's CodeXL, we report that all kernels have an occupancy of 70% and the main bottleneck on the number of waves per SIMD unit seems to be the number of scalar registers (96 are used, which corresponds to a score of 8/10; below 81 would be ideal). The tool also shows that the implementation would highly benefit from more local memory and more vector register usage (where 4–23 vector registers are used, depending on the kernel). A prime example of a kernel which could still benefit from local memory usage would be the density kernel (25.81% of run time, the most time-consuming kernel), where locality could be a way to tile memory. This is not to say that we don't already employ local memory in some places; examples include the velocity kernel (where we

highlight the increased granularity of atomic additions needed to count the number of walls, as seen in [33]).

In order to quantify if there is a data transfer bottleneck, we first remark how the overlap between compute and data transfer works: one has two different queues, one for data transfer, one for kernel execution, and using events one triggers data transfer upon completion of the partial sums kernel. Unfortunately, to allow for overlap, the enqueueing of data transfers needs to be nonblocking. After enqueueing some kernels, it is important to wait for the data transfers to complete (to ensure the sum of partial sums isn't summing over garbage). Since the waiting time will also include waiting for compute kernels to finish (again enqueueing kernels is a nonblocking operation) we estimate the time taken by data transfer to roughly correspond to the difference between waiting time and total kernel execution time. Comparing to the run time reveals that data transfer is only a bottleneck in low resolution boxes.

Significant loss of precision need not occur from single precision, though in OpenCL division and square root operations do not generally apply correct rounding as prescribed by the IEEE754 specification. For this graphics card (and implementation), a JIT compiler flag can enable it by passing the option *-cl-fp32-correctly-rounded-divide-sqrt* for single precision arithmetic only. Since this option is not available for double precision, we compare the sequential and parallel implementations for the two aforementioned diagnostic quantities, either using single or double precision. To do so we evolve several boxes with the described settings using the same initial conditions across the board (generated by the single precision code, to avoid hamstringing the single precision version at initial time steps due to typecasting rounding errors).
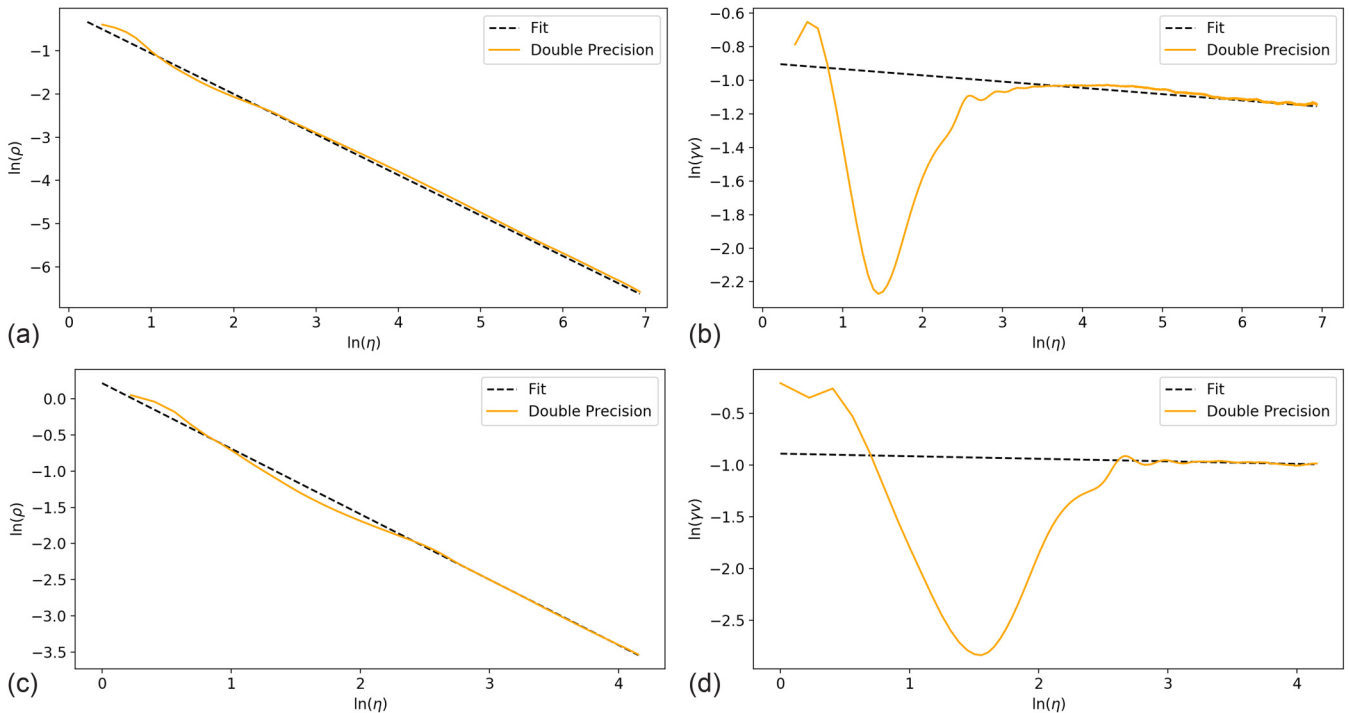

FIG. 3. Evolution of the density (ρ, left panels) and the velocity (γv, right panels), for 2048^2 and 128^3 box simulations (top and bottom panels, respectively), showcasing the expected scaling behavior.

In both the double and the single precision case, the differences between the parallel and sequential versions seem to be negligible after the early time steps, once the wall networks have eased the "numerical" initial conditions in the box and are approaching the scaling solution. Both errors seem bound by the maximum precision specified by their data types (for this specific machine and as dictated by the *FLT_DIG* and *DBL_DIG* macros) at latter time steps. This can be seen in Fig. 2. Note that the single precision case tends to incur a much larger error during the initial time steps.

As a final validation, we use sets of five single and double precision runs of 2048^2 and 128^3 boxes to calculate the scaling exponents defined in Eq. (9). The same set of five fixed seeds is used both in single and in double precision. The scaling exponents are calculated using a linear fit and ignoring the early part of the simulations (whose dynamics is still dominated by the initial conditions). The calculated exponents are listed in Table I, and are in agreement with previous simulations of boxes of these sizes with CPU versions of the code [27,28]. The listed uncertainties are statistical, from the average of each set of five runs (this is the relevant comparison here); additional systematic uncertainties in these diagnostics are discussed in [8]. Figure 3 depicts the evolution of the density and the velocity, illustrating the expected approach to the scaling behavior.

IV. CONCLUSION

We have ported a previous sequential code based on the PRS algorithm to a parallel OpenCL-based implementation, specifically optimized to the GPU used. This highlights the point that even with a consumer grade graphics card reasonable speedups are to be expected, provided a large enough box size is used. We also investigated the possible loss of precision. The fastest version but with higher error corresponds to the single precision version with compiler flag *-cl-fp32-correctly-rounded-divide-sqrt*. Both the single and double precision version yield consistent results for the scaling diagnostics (keeping in mind that larger boxes yield better results).

The bottleneck at larger box sizes will be the amount of memory available to the graphics card; this might be lessened by reducing memory usage. Changing graphics card will require reoptimization of the code. Further ongoing work includes optimization for central processing units (as OpenCL guarantees portability of code—minor implementation differences aside—but not optimized execution for all types of devices) and a comparison between the parallel codes on the GPU and on the CPU. After these further validations and verifications we expect that the GPU domain wall codes may be used for generating large sets of production runs for astrophysical exploitation.

ACKNOWLEDGMENTS

[1] T. W. B. Kibble, J. Phys. A **9**, 1387 (1976).

[2] A. Vilenkin and E. P. S. Shellard, *Cosmic Strings and Other Topological Defects* (Cambridge University Press, Cambridge, UK, 1994).

[3] T. W. B. Kibble, Nucl. Phys. B **252**, 227 (1985).

[4] C. J. A. P. Martins and E. P. S. Shellard, Phys. Rev. D **54**, 2535 (1996).

[5] C. J. A. P. Martins, *Defect Evolution in Cosmology and Condensed Matter: Quantitative Analysis with the Velocity-Dependent One-Scale Model* (Springer, Berlin, 2016).

[6] W. H. Press, B. S. Ryden, and D. N. Spergel, Astrophys. J. **347**, 590 (1989).

[7] C. J. A. P. Martins, I. Y. Rybak, A. Avgoustidis, and E. P. S. Shellard, Phys. Rev. D **93**, 043534 (2016).

[8] C. J. A. P. Martins, I. Y. Rybak, A. Avgoustidis, and E. P. S. Shellard, Phys. Rev. D **94**, 116017 (2016).

[9] J. Briggs, S. J. Pennycook, E. P. S. Shellard, C. J. A. P. Martins, M. Woodacre, and K. Feind, *Unveiling the Early Universe: Optimizing Cosmology Workloads for Intel Xeon Phi Coprocessors in an SGI UV2000 System* (SGI/Intel White Paper, 2014).

[10] D. P. Bennett and F. R. Bouchet, Phys. Rev. D **41**, 2408 (1990).

[11] B. Allen and E. P. S. Shellard, Phys. Rev. Lett. **64**, 119 (1990).

[12] C. J. A. P. Martins and E. P. S. Shellard, Phys. Rev. D **73**, 043515 (2006).

[13] K. D. Olum and V. Vanchurin, Phys. Rev. D **75**, 063521 (2007).

[14] J. J. Blanco-Pillado, K. D. Olum, and B. Shlaer, Phys. Rev. D **83**, 083514 (2011).

[15] J. N. Moore, E. P. S. Shellard, and C. J. A. P. Martins, Phys. Rev. D **65**, 023503 (2001).

[16] T. Hiramatsu, Y. Sendouda, K. Takahashi, D. Yamauchi, and C.-M. Yoo, Phys. Rev. D **88**, 085021 (2013).

[17] M. Hindmarsh, J. Lizarraga, J. Urrestilla, D. Daverio, and M. Kunz, Phys. Rev. D **96**, 023525 (2017).

[18] A. Lopez-Eiguren, J. Urrestilla, and A. Achúcarro, JCAP **01** (2017) 020.

[19] A. Achucarro, A. Avgoustidis, A. M. M. Leite, A. Lopez-Eiguren, C. J. A. P. Martins, A. S. Nunes, and J. Urrestilla, Phys. Rev. D **89**, 063503 (2014).

[20] P. McGraw, Phys. Rev. D **57**, 3317 (1998).

[21] M. Hindmarsh, K. Rummukainen, and D. J. Weir, Phys. Rev. D **95**, 063520 (2017).

[22] P. A. R. Ade *et al.* (Planck Collaboration), Astron. Astrophys. **571**, A25 (2014).

[23] J. Aasi *et al.* (VIRGO, LIGO Scientific), Phys. Rev. Lett. **112**, 131101 (2014).

[24] F. Finelli *et al.* (CORE Collaboration), arXiv:1612.08270 [astro-ph.CO].

[25] P. Binétruy, A. Bohé, C. Caprini, and J.-F. Dufaux, JCAP 06 (2012) 027.

[26] B. S. Ryden, Astrophys. J. **333**, L41 (1988).

[27] A. M. M. Leite and C. J. A. P. Martins, Phys. Rev. D **84**, 103523 (2011).

[28] J. R. C. C. C. Correia, I. S. C. R. Leite, and C. J. A. P. Martins, Phys. Rev. D **90**, 023521 (2014).

[29] A. Munshi, OpenCL 1.2 Specification, Tech. Rep., 2012 (unpublished).

[30] M. Scarpino, *OpenCL in Action* (Manning Publications, Shelter Island, New York, 2011), pp. 1–458.

[31] AMD Graphics Core Next Architecture, Tech. Rep., 2012 (unpublished).

[32] AMD OpenCL Optimisation Guide, Tech. Rep., 2014 (unpublished).

[33] S. OpenCL, Performance of atomics, 2013, http://simpleopencl. blogspot.pt/2013/04/performance-of-atomics-atomics-in.html.